Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 3 (2011) 1283–1295

Procedia Computer Science

www.elsevier.com/locate/procedia

WCIT 2010

A tool for evaluating event based middleware

Cosmina Ivan[a] *, Vasile Dadarlat[a]

[a]*Faculty of Automation and Computer Science, Technical University of Cluj Napoca* 26-28, Baritiu„Cluj-Napoca , ROMANIA

Abstract

Notifications services are the middleware that provides information dissemination and other important features upon which publish/subscribe systems rely. Several commercial products, open source and research projects implements notification services as distributed event brokers that considerably differ each other in features and QoS that they provide. Great research efforts are concentrated in areas related to information dissemination, efficient routing algorithms, optimal location of filters for advertisements and/or subscriptions, and various optimization of resources use, scalability, fail tolerance, etc.
The own nature of the notification services as distributed, large scale, big amount of concurrent messages processed, etc., as well as the diversity and complexity of the facets involved in their implementations, impact negatively on the analysis and evaluation of their behavior while they are in execution. There exist few solutions that solve this problem but they are mainly proprietary and focus on single or only partial aspects of the behavior. We presume that our solution would significantly help to the research, development and tuning of these modern distributed systems.
The main motivation of this thesis is to enable the analysis of notification services implemented as distributed event brokers while they are running. This work introduces a notification service-independent analysis framework in terms of design , implementation and prototype evaluation , which allows online behavior analysis based on streamed observations, metrics and visual representations of them.

Keywords: publish-subscribe, AOP, notification services

1. Introduction

The new scale of the distributed systems due to Internet has given rise to applications based on the dissemination of very high volumes of information. Applications like news services, stock-quote exchange, remote monitoring are traditional examples of this kind of applications. These applications in addition to the processing of big amounts of data is characterized by loosely-coupled components that generally requires the execution of actions as response of certain happenings events. In publish /subscribe systems, producers publish messages or events and these are delivered to subscribers according to their interests expressed previously as subscriptions. This allows to disconnect producers of consumers communicating them in an indirect and asynchronous way. The event notification services are systems that should assure that the published notifications or messages arrive at the correct subscribers. A notification service could be implemented as a centralized service. But usually they are implemented as a distributed event brokers network that collaborate to perform the necessary tasks getting support for large scale systems.

Publish/subscribe-based messaging systems are used increasingly often as a communication mechanism in data-oriented web applications such as Web 2.0 applications, social networks, online auctions and information dissemination applications to name just a few .Moreover, the publish/subscribe paradigm is part of major technology

* Cosmina Ivan. Tel.:0040744223652; fax:0040264438003
E-mail address:cosmina.ivan@yahoo.com

1877-0509
doi:10.1016/j.procs.2011.01.005

domains including Enterprise Service Bus, Enterprise Application Integration, Service-Oriented Architecture and Event-Driven Architecture.

Nowadays there are different research projects about notification services (REBECA, Siena, Hermes, Jedi, Scribe). Efforts are concentrated in areas related to information dissemination, efficient routing algorithms, optimal location of filters, optimization of resources use, scalability, fail tolerance, etc. Commercial products like WebSphereMQ , TIB-Rendezvous, FioranoMQ or JMS specification [] ,and its open source implementations OpenJMS ,JBoss,ActiveMQ, Joram offer different kinds of routing algorithms, persistence mechanisms, messages type support, subscription languages, transactions support, scalability, etc. Each notification service implementation has a different combination of this mechanisms and improvements. Therefore, it is possible to get different benefits and specific behaviors between one notification service and others using it in certain types of applications or specific configurations. The own nature of the notification services: distributed, large scale, big amount of concurrent messages processed, etc., as well as the diversity and complexity of the facets involved in their implementations, impact negatively on the analysis and evaluation of their behavior *while they are in execution.*

The solutions to this problem are generally based on the processing of low-level abstraction data collected during the execution but available afterwards. This data is usually taken with the semantic of the source code and ad-hoc built for each implementation. Some other few solutions exist that cover this problematic but they are mainly proprietary and focus on single partial aspects of the behavior. By being able to analyze notification services independently of their implementation would make possible the comparisons between them. With the analysis and measures of detailed internal mechanisms it is possible to focalize individual aspects of the behavior, like time consumption associated to the persistence of messages, the efficiency of the routing algorithms, or the specific time consumed to match subscriptions, etc.

This paper is organized as follows. Section 2 shows a background about pub/sub systems, its main characteristics and behavior as well as analysis of related work and the proposed approach is explained in detail. Section 3 starts analyzing requirements and main issues that have to be considered for these three building blocks, communication and components management are analyzed and putting it all together in a high level architecture and maps the architecture into the design. Analyzed requirements are solved with design solutions and implementations alternatives are presented for some components. Section 4 describes the main decisions taken in order to develop a prototype implementation of the architecture. And Section 5 concludes this work and sketches possible enhancements and areas of potential application or future work.

2. The problem domain and conceptual design

2.1. Publish – subscribe systems

Notification services are a key element in publish/subscribe systems. In these systems, loosely-coupled clients could communicate with each other through messages in an asynchronous fashion. The clients could assume two different roles, publisher or subscriber. Subscribers are message consumers that express particular interests on messages. These interests are expressed as subscriptions. Publishers produce data that is disseminated as messages. These messages are delivered by a notification service to the subscribers according to their expressed interests. Notifications services are the middleware that provides information dissemination and other important features upon which pub/sub systems rely. The interaction between the clients and the notification service is determined by a basic scheme. Subscribers have a way to express their interests (subscribe/unsubscribe). Publishers have a way to disseminate messages (publish). Finally, the notification service is in charge of delivering the published messages to corresponding subscribers (notify). In spite of this simple interaction, the pub/sub systems could adopt a broader variety of operation alternatives and this flexibility makes them sometimes complex but it allows their adoption to the specific requirements of different applications. One of the fundamental characteristics that allow to classify pub/sub systems is the addressing model that determines the correspondence between publications and subscriptions.

Some of the typical addressing models are[1,2]:

- *channel-based:* It was the addressing model adopted by the first generation of pub/sub systems. A channel is used to communicate producers with consumers. Channels allow efficient information dissemination but the expressiveness is rather limited.
- *subject-based:* Under this category publishers label each message with a subject name. Consumers subscribe to names and messages matching with them are then delivered. Subjects names could be a hierarchy of

elements (i.e. element.subelement.subsubelement). Different wildcards could be used to express subscriptions among the hierarchical relationship of subjects.

- *content-based:* It introduces a subscription scheme based on properties of the considered messages. The subscriptions are expressed by name-value pairs of properties and usually combined using logical operators like and, or, etc. This forms a complex subscription pattern which is usually translated into a filter that is evaluated against incoming messages to discovery interested subscribers.

A notification service could be implemented as a centralized service, but also as a distributed event brokers network that collaborate to perform the necessary tasks. Different addressing models and topology adoption implies a different algorithms and mechanisms set that have to be implemented in a notification service to support them. Additionally, the notification services could vary in their implementations offering or not and in different ways an important diversity of features and quality of services (messages persistence, transactions, fault tolerance, security, routing algorithms, efficient dissemination, etc.).

The behavior of a notification service is understood by those mechanisms that it carries out to ensure the correct interchange of messages according to the pub/sub paradigm. For each different extra set of features and quality of services offered different mechanisms are implemented. There are two basic mechanisms that are present in all implementations of notification services[1,3]:

- By each message that is published the notification service has to determine which subscriptions match and which are the corresponding consumers. This mechanism is called *matching.*
- When the matching process is done, the message that was published has to be delivery to the subscribers. The delivery process to the corresponding subscribers is called *routing.*

Each of these two basic mechanisms has several variants according to different addressing models and topology used , and generate a real diversity in terms of implementations.[8]

2.2. Behaviour analyzing tools

Tracing and profiling is a traditional approach to debug and analyze the execution of general systems and also it is usually applied to notification services. Additionally, big ad-hoc text files are generated registering all events that happen during the execution. This mechanism is easy to implement because it is only necessary add extra source code finding first the right places and then modifying the code to write the relevant data into a text file. All the execution flow captured is stored in these files using a very low level of abstraction, at source code level. This feature gives them rich details that make log files suitable to respond to very specific questions. Consequently, their size grows quickly and they become enormous text blocks that are very difficult to manipulate and to analyze manually. Since tracing information are taken at source code level, records that are generated in this approach depends on each implementation. People that want to analyze their behavior have to know and interpret the source code or internal structure of the implementation. In the case of distributed systems, the analysis of the behavior from log files is still more complex.

Black- box analysis .Another usual approach to analyze the behavior of the notification services consists of registering information outside the limits of the same one. Basically, data are observed and registered at the clients, when messages are published and/or received . This approach is particularly spread between JMS implementations. JMS API defines a standard way to communicate clients with notification services. Therefore, it allows to reuse custom clients in a simulated injection of messages for different notification services. The main idea of this approach is to register the data of a message before it enters the notification service and after it arrives at the subscribers. By taking timestamps at each of these points in time it is possible to calculate the time that a message took from its publishing to its delivery. By doing a controlled injection of messages and registering these data, values that measure some aspects of the behavior can be obtained.. This higher abstraction level than log files allows to understand the values with independence from the source code and let make comparisons between different notification services. The major problem with this approach is that it cannot measure the details of the behavior while the message is processed inside the notification service.[4]

Our approach proposes to observe notification services while running and take data about relevant processing events that are applied to each message. Then, this little and atomic information is composed in a flexible way to get values that allow measuring different behavior aspects. These values are associated to the definition of metrics that allow a high-level abstraction analysis of the behavior.

In order to allow near real time analysis of big amount of data, different visualization techniques can be configured to represent these values and visualize the behavior of the notification service observed. We propose the the following major building blocks for this technique :

- *behavior observation:* abstract and NS-independent data records are produced by notification service observation. This data is not statically stored, it flows towards the next piece.
- *metrics composition:* the data records that proceed from observation are gathered and composed to get measures with a higher abstraction level than the primitive events observed. This compositions are defined by the user as metrics of analysis.
- *behavior analysis:* the values that come from the metrics composition are visualized in different ways according to the user preferences.

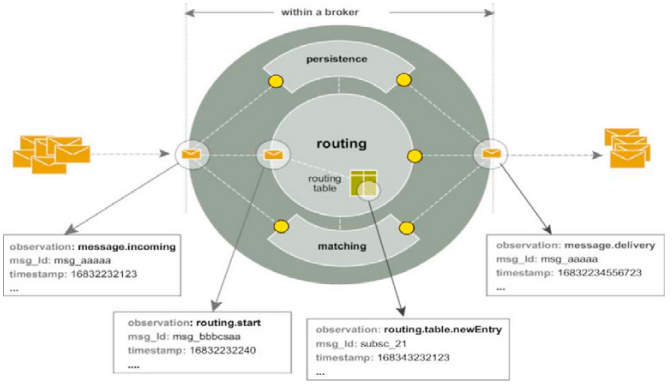

Fig.2.: Taking observations from relevant execution points.

There are few solutions that cover this area but all of them are proprietary of each particular implementation and in general they are restricted to cover partial aspects of internal behavior of notification services. The commercial products in general have configuration and monitoring mechanism, that fundamentally allows to make tuning and report or analyze the use of resources. But they are partial solutions and focus on resources consumed rather than in internal behavior.

There are also some works based on the JMS specification, but these tools are generally based on the activity that take place outside the event broker and they can't measure the influence of each individual aspect of the behavior in the processing of the messages. It does not exist a well-known specification of metrics to be able to evaluate notification services. There are also some works related to the visualization of complex systems [11] but most of them are based on static data that proceed from logs files or databases to support post mortem analysis.

The main focus of this work is on developing an analysis framework for event based middleware named notification services independently of their implementations. This analysis framework is based on the definition of observations, behavior metrics and visualization representations of them. The main goal is to allow near real-time analysis while the notification service is running (online analysis), but different kind of post-mortem analysis were contemplated and allowed with the same solution. Another accomplishment of our work is the design and the implementation of this analysis framework as a prototype.

3. Analysis and design of the proposed architecture

Initially, the proposed approach consists of collecting data that represent the internal behavior of a notification service. In order to get this data it is necessary to know the internal organization of the source code at each event broker. Then it should be observed the mechanisms that act on the messages The main idea is "to follow the messages" within event broker. At the same time all relevant execution points or state changes that take place in the internal structure are registered while messages are being processed (Figure 1) .

By each one of these execution points or state changes a record is elaborated taking descriptive data according to the observed behavior. These records will be called *observations.*

For all *observations,* the following data should be considered:

- *observation type:* It allows to identify the *observation* and associate the data that it contains with some aspects of the notification service behavior.
- *message identification:* It is an identifier assigned by the notification service for each message. It is used to gather observations taken on the same message. (for correlation purpose)
- *timestamp:* It is the point in time when the observation was taken.

Once grouping all the observations that affect a certain message, it is possible to reconstruct the "path" that has followed the message through the event broker. With this "path", it is possible to obtain the detailed behavior that the event broker carried out when processing it, and the exactly moment when it did it.

The architecture main building blocks must detect and capture the behavior of the notification service. In order to do that, it observes the flow of messages within each event broker and the mechanisms and state changes that occur in its internal structures while messages are processed. By observing behavior, related information is captured and written in the *observation* data structure. The main tasks that this building block carries out are:

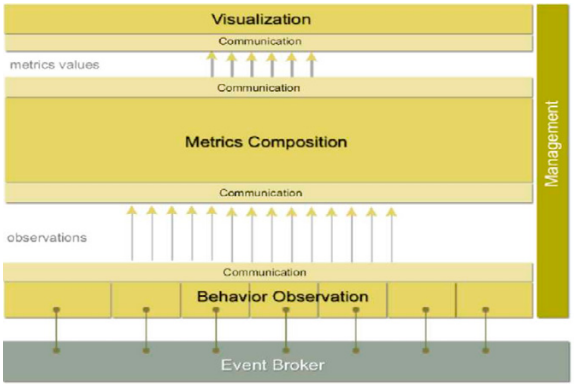

Fig.3 The high level architecture

1. *Detect the behavior:* It should intercept and capture data from the source code executed at each event broker when messages are processed.
2. *Generate observations:* With the information captured about observed behavior, a record that describes this behavior and the corresponding features of the processed message is generated.
3. *Send observations:* Data records with behavior description are not stored statically. Instead they become streams towards another building block that uses them to compose metrics.

At each relevant point where the behavior is observed, an *observation* instance is produced. In the context of this work, the *observation* is the minimum unit of information about the behavior of a notification service. Each observation describes in detail *what, when* and *where* the behavior that it describes was observed. One of the main requirements that must solve this building block is to manage the independence from notification services. The goal is to allow a high level and homogenous analysis of the behavior of notification services, whatever their implementation would be.

The observation and capture of the behavior depend on each notification service since it should be made at source code level. It is important that the structure of the *observations* stays independent of the implementation to solve the abstraction of the data towards the other pieces. Therefore, if the *observation* is abstract, the piece that composes the metrics receives information whose semantics is associated to the behavior of a notification service, independently of how it is implemented.

In order to do that, the idea is to define initially which observations we need to take, independently of how we will take them. Then, we have to study the internal organization of the source code of the event broker and associate each observation with some portion of code. The point of execution where the behavior is *detected* is, undoubtedly, owned by each notification service but, it is desirable that *generation* and *sending* of observations could be reusable and useful for more than one specific notification service.

This building block has the goal to manipulate *observations* in order to compose metrics. The output of metrics composition is a flow of values associated to each metric that is computed. In order to get these values, compositions are carried out over observations according to an external metric definition provided by the users. This definition is a set of grouping, transformations, aggregations and others operations that are applied on the attributes of some selected *observations.*

As it was mentioned , the online analysis requires the near real-time visualization of the values associated to metrics. A visual component will receive at certain short time intervals new values and the graphical representation will be updated. Therefore, it will be necessary to contemplate that each one of the implemented techniques can dynamically vary, for instance, the scale and other characteristics that depend on the global nature of the data to be displayed.

In the case of the offline visualization, the values to be displayed are also pushed but now from a repository. The repository can be explored previously and it can be determined the most suitable scale or other characteristics before the values begin to arrive.The users will have a visualization panel in where it will be able to analyze the behavior of notification service from defined metrics and the graphical representation selected for each one.

The visualization panel will be the visualization client and it must act like a container of the visual components that implements the visualization techniques. All the issues related to layout, size of the components, location, etc.

are responsibility of this client. On the other hand, some necessary management tasks exist so that the user can control in different ways the visualization of the metrics, which also must be considered by this client.

During the online analysis, situations that are detected can require that the user needs to analyze other metrics, or that current metrics must be visualized in a different way. We must consider for these cases the possibility of dynamically change the metric that are visualized and/or the techniques that visualize them[11].

The architecture that sets out divides in three different and independent layers. The bottom layer deal with the observation of the behavior and acts over the event brokers of notification service that will be analyzed. The next layer conducts the operations necessary to compose metric from the definitions of the same ones. The upper layer is the visualization client that contains the visual components that represent graphically the metric that are composed. These layers are communicated to each other trough the network with the objective to make flow the data and in order to get online visualization. *Observations* flow between bottom and middle layer and between the middle and the upper layer flow values associated to metrics. A last layer cross the previous ones and will be responsible for management tasks that are needed to operate the capture, transformation and visualization of the data as well as the orchestration of components.

In order to made flow the *observations* between behavior observation layer and metrics composition layer and then made flow the metrics values computed towards visualization, the three main layers have to be communicated each other through the network. Two wished characteristics that the communication model that will be selected has to accomplish are:

- *Push based:* The observations are generated and sent as the messages flow within each event to broker. The groupings and transformations are due to make as the necessary observations are arriving at the metrics composition layer. Whenever they are obtained new metrics associated values are sent to the visual components that must update their representation immediately. The communication is asynchronous in both cases and in only one direction always initiated by the lower layers. On the other hand, an observation is used to compose one or more metrics and a metric is visualized by one or more visual components. In both cases the communication is 1-to-N. It is clear that Push technology is more adequate than the traditional pull based model for this cases . Using pull based communication, the superior layers would have to stay doing data polling with consequent performance and bandwidth costs.
- *Firewalls:* The bottom layer acts on each event to broker. These event brokers can be distributed in different geographic locations and to belong to different organizations. Moreover, could be multiple visualization clients also distributed. It should be considered the restrictions imposed by the existence of firewalls like instruments of security policies in several organizations. It is desirable the use of communication protocols like HTTP to communicate the layers, which should allow to cross even the most restrictive types of firewalls.

The objective of the management layer is to carry out all those necessary tasks for the administration, configuration and orchestration of the components that integrate this solution. The user should have to do some tasks related with metrics definition, configuration of graphical representations for them, reproduction control at offline analysis, among others. Hence, this layer is responsible of components orchestration, necessary tasks to put them in operation in order to bring an analysis oriented application.

We will describe shortly the proposed functionality for each layer of our architecture.

The behavior observation layer is the bottom layer of the proposed architecture. This layer observes the behavior, generates the observations and sends them to the metrics composition layer. For each of the three main tasks that carries out this layer there different dependency levels with the notification service are observed:

- behavior detection and capture: The detection and capture of the behavior is totally dependent on the implementation of the notification service under analysis. The source code must be examined and some executions points have to be intercepted .
- observations generation: Once the desired behavior has been detected and captured, an observation has to be generated. The dependency on the implementation is relative at this point and depends on how the capture was made. Observations generation must ensure that the created observation is independent of the implementation. The upper layer will receive generic data structures that represent observations and it always knows how to treat them.
- observations sending: The observations sending is independent of the observation generation. It depends on the chosen communication model.

As it has been analyzed in the previous chapter, one of the main requirements that must fulfil this layer is to avoid influencing the behaviour and the performance of the notification service observed. The behaviour

observation layer contains the following components :a first sub-layer detects the behaviour and generates the different observations. This sub-layer is, at the most, the only layer that will be necessary to be modified in case of changing the notification service observed. A second sub-layer operates on that sub-layer to enable or disable the generations of observations. A third one separates the tasks referred to the observations sending and the communication with the metrics composition layer. This last sub-layer let decouple the observation of the behaviour from the communication allowing the selection of different communication models and/or the sending frequencies. Finally, a sub-layer of management crosses to the previous ones implementing the operations necessary to control the mechanisms exposed in previous sub-layers. In the next subsections, these components and design issues are explained.

A notification service executes a sequence of actions to process a message since it is received until it is delivered. The source code that implements this behavior is scattered among different classes. We must intercept and observe some of those execution points and capture data about the message and the applied behavior. A different approach to avoid these disadvantages is to use Aspect Oriented Programming (AOP) .With AOP [9,10], logic associated with an specific concern could be developed in a modular way and weaved into to program statically (compilation-time) or dynamically (load-time, run-time). AOP bring us a suitable solution that lets simple maintainability and avoids the modification of the source code of the notification service.

The sub-layer of observations sending sends the observations to the metrics composition layer. On one hand, it offers to the generation sub-layer an abstraction of the selected communication. On the other hand, it implements different alternatives to adapt the sending frequency of the generated *observations.* Some of these alternatives include:

- *Online sending:* In this case, the observations are being sent one at a time after they are observed/generated.
- *Offline sending:* The generated observations are accumulated without being sent to the upper layer. An order from the management forces the delivery of all collected observations.
- *Time intervals:* In this case the observations are accumulated but they are sent at each certain predetermined time interval.
- *Amount fixed:* Like in the previous alternative, the observations are sent when an specific amount of them has been accumulated.
- *Package Size:* The observations are collected until the size of a package has been reached. It can be used to optimize the use of the bandwidth of the network.

All these possibilities allow an appropriate decision considering the best use of the involved resources (i.e. network, memory, CPU). Management in this layer carries out the operations needed to control the generation and sending of *observations.* It acts in a remote way regarding the main management and in a distributed way if more than one event broker exists. It has to listen to orders that can arrive from the main management. The two main tasks that management carries out in this layer are:

- *Enable/Disable observations*: By means of this operation, the generation of certain *observations* is controlled having been able to (dynamically) enable or disable the capture and generation of *observations* of certain facets of the behavior.
- *Sending frecuency control:* The objective of this task is to be able to (dynamically) vary the mode of observations sending.

Both tasks depend on how the detection and generation of observations is implemented. The following section analyzes implementation alternatives and how each of them affects these tasks.

The metrics composition layer is the middle layer in the proposed architecture. This layer receives a high volume of data flowing from the behavior observation layer. The goal of this layer is to reduce this data flow to different metric values according to user-defined definition. As it was analyzed in the previous chapter, compositions should be produced ensuring low-latency to let near real-time (online) analysis. Additionally in this layer, observations an metrics have to be stored to allow different kinds of post-mortem (offline & static) analysis.

- Metrics Compositors**:** Compositors are the core of this layer. They are a set of mechanisms that should process observations in a suitable way in order to produce/calculate values associated with metrics defined by the users.
- Repository**:** It stores observations that proceed from the bottom layer as well as the metrics values produced by compositors in this layer. This stored data would be used to support post-mortem analysis.
- Metrics Definition**:** Metrics definitions are a set of high-level descriptions that configure compositors in order to reduce observations to specific metrics values. Each compositor is associated with a specific metric definition.

Different metrics will have different attributes according to the transformations that have been done as well as the collection of observations from which it was composed. The representation of these attributes have to be independent of the visualization layer. Metric definitions point out to compositors how their output data have to be represented in order to ensure this necessary independence. The output data from compositors is a concrete instance of a particular metric.The repository is a persistence system that stores observations and calculated metrics values. It can be accessed issuing specific queries after the online analysis is done. The election of a particular technology for it depends on data representation selected and how these stored data would be structured to facilitate different kinds of post-mortem analysis.

The visualization layer receives a flow of values and must visualize it using different visualization techniques. The idea is to offer the user a set of techniques and let him to decide which of these techniques are selected to visualize each metric. there are multiples visualization techniques to represent information. Each visualization technique will be implemented in this layer by an independent and reusable visualization component. Each of these components receives a data flow as input and has to represent it graphically somehow. Finally, these components are located in a visualization panel, like a dashboard, to support online analysis. A visualization component is a piece of software that implements a visualization technique. It receives a flow of metrics values and knows how to graphically represent it. The goal of these components is to avoid custom implementation of data transformation and visualization code to get it graphically. These components are decoupled from metrics semantics and each of them could be reused to graphically represent more than one technique. New components or visualization techniques could be added without the need to modifying data representation of metrics neither another visualization component.

Metrics instances are delivered to the visualization components in an asynchronously fashion. Once a metrics instance is received several values from it have to be extracted. To graphically represent these values it is necessary to interpret them in some way. For example, if the implemented technique is a vumeter, we need to know which attribute from the metrics instance is the value that "moves the needle". If the technique is a pie chart it is necessary to know which are the attributes associated to each sector. Therefore, the elements of a visualization techniques have to be associated to the attributes of metric instances. Data extraction mechanisms are initialized with a set of initial parameters that let them extract and interpret values from metrics instances.. Once a metric instance is received the graphic representation is updated with the extracted values. Since the whole data is not known in advance visualization components have to deal with dynamic updates of scales. Additionally, each visualization component could offer to the user a set of mechanisms to customize its visualization technique. This customization depends on each technique and how it is implemented.

In order to be able to tie a metric to a visualization technique it is necessary to map between these two elements. We know that each visualization component does not know how to interpret a metric but it has a mechanism to extract the data from the flow of metrics instances that it receives. This mechanism must be parameterized based on the attributes of metrics and the elements that conform the visualization technique that the component implements.For the simplest case it is necessary to assign an attribute from the metric as the input data of the vumeter. Then, when a new metrics instance arrives the value of this attribute would move the usually needle. In the case of a bar chart for instance, it is possible to make different mappings between the metric instance and its elements. An example could be to assign a measure as the value to represent by Y axis and the value of a dimension as the labels of data categories. Then, constant values like the dimension or measures names could be assigned as corresponding axis names. Thus, when receiving the first metric instance, the axis and labels are initialized and a first series of data is graphically represented. The extracting data mechanism knows which are the attributes that contains the values to represent and which are the elements where to display them. With the next incoming instances, the values represented will be updated in the graphic.

Since the visualization panel has to run in a web browser, its implementation as well as the implementation of visualizations components have to be carried out applying web client-side technologies. There are different kinds of these technologies and each of them has pros and cons for our purpose. Some of them are: • *DHTML-Ajax, Java Applets ,Scalable Vector Graphics (SVG), Macromedia Flash, Web3D standards(VRML/X3D).* Each of these technologies enumerated has different features, pros and cons but note that the selection of each one for panel or visualization components implementation generate dependency consequences on each other. The suitable situation should be to get independence to integrate in a same panel visualization components implemented in any of these available technologies.

The communication of the components is a central part of the architecture and determines dependencies in the three exposed layers previously. There are at least two alternatives that are analyzed in the following subsections.

One alternative is the traditional centralized approach. There at least three main problems with this approach. A first problem relies on the limitation of HTTP to implement push-based communication which is commonly based on sockets. HTTP connections are assumed to be short-lived and based on request/reply paradigm. There some solutions to this problem based on having as the response of a request an infinitely large number of responses keeping alive the connection. Two different situations for the same problem are: a) the sending from aspects to the metrics layer (client to server) and b) the sending from metrics to visualization (server to client). A second problem is the scalability of this solution. We know that the volume of messages flow in the notification service under observation could be very large, then the flow of observations is at least like that volume.

Another option is to communicate the involved components using the pub/sub paradigm. In this case the components interchange messages through a notification service. Each component could be a publisher, a subscriber or both.Compositors acts as subscriber and publishers. The repository subscribes to all, because it is interested in observations as well as metrics instances and they are stored simultaneously to metrics composition process (d). Finally, each visualization component subscribes specific metrics instances according to the metric which it have to represent .Related tasks regarding the routing of observations and metrics values, scalability, reliability, fault-tolerance, etc. are delegated to the pub/sub middleware. New components in the three layers (aspects, compositors, visualization components) could be added or removed dynamically and independently of each others. Adopting this approach naturally brings an asynchronous, push-based, N-to-N communication model that can be highly scaled. The components are in a loosely-coupled fashion and quality of services issues are delegated completely to the notification service. Due to all these advantages we decided to adopt it as communication mean.

4.Prototype implementation

Before beginning the development of the three building-blocks, it is necessary to take some initial decisions. These decisions involve the selection of a pub/sub system to communicate the different layers and the election of how to represent the information that flows among them. Finally, we have to select a suitable repository to be able to store the necessary data to support post-mortem analysis as well as to store the information that arises from metric definitions and other configuration and administration tasks.

The Behavior observation layer has been implemented with AOP to observe, capture and generate *observations.* Additionally, classes for management and sending observations were implemented. For the prototype implementation we have chosen AspectJ as language. AspectJ assures reduced overhead and there are spread tools that support the development. It would be possible to replace these aspects in the future without modifying too much the rest of the classes if dynamic weaving it is preferred to use.[10] Figure 4 shows the classes involved in the development of this layer. The ObservationManager class implements management tasks necessary in this layer. This class has a PushletSubscriber to subscribe to special messages that come from test runs management. These messages could be to enable or disable certain observations generation and to change the sending frequency strategy. The enable and disable mechanism depend on the type weaving used. AspectJ has static weaving and is not possible to dynamically add or remove aspects. These operations have been implemented according [] using a static boolean flag because it allows to optimize performance overhead when the aspects are disabled.

The Metric composition layer involves the definition of metrics and the generation of compositors from them. The definition of metrics implements the data warehouse metaphor by means of dimensions and measures description from attributes of the observations. Compositors has been implemented adapting and extending filters for notification service proposed in [8]. These filters are generated from the metrics definition parsing. Additionally management tasks were implemented to control these compositors which are necessary for components orchestration. In the following sections the details of the implementation of these elements are presented. Components were used for the implementation of compositors. These components are a set of filters that can be incorporated in different ways to a notification service being able to manipulate data streams. These filters are generated on a set of diverse primitive that allow to configure in a flexible way different event compositions. For our particular case we have extended a filter of the type aggregator . Basically, this filter allows to subscribe to a flow of events and to accumulate these events while a condition of completeness is not fulfilled.

The visualization layer has been implemented to be able to be executed within a web browser. The visualization panel is a web document that is generated according to the selections of the user. Third party visualization components can be easily integrated in this panel. Following subsections explain details of the implementation of visualization panel as well as the support for the integration of visualization techniques integration. The implementation of the visualization panel determines in part how visualization components could

be integrated and the technologies in which they could been developed. The visualization panel is responsible to present the corresponding visualization techniques according to user-defined selection as well as to tie each of them with the corresponding flow of metrics values. The metrics definition as well as the mappings between them and the visualization components have to be recovered from the repository according to the test run parameter that receive this panel. The visualization panel is implemented as a web document that is loaded in the browser according to an HTTP request parameterized with the context of the visualization (test run). This document is dynamically generated by a server-side script that is explained below. Taking advantage of *iframe* element from HTML, that let embed externals web documents in a same one, it is possible to integrate visualization components developed with different web technologies. In this way, each visualization component will be invoked by setting the *src* attribute on each *iframe* element. This new request will be responded by a new server-side script

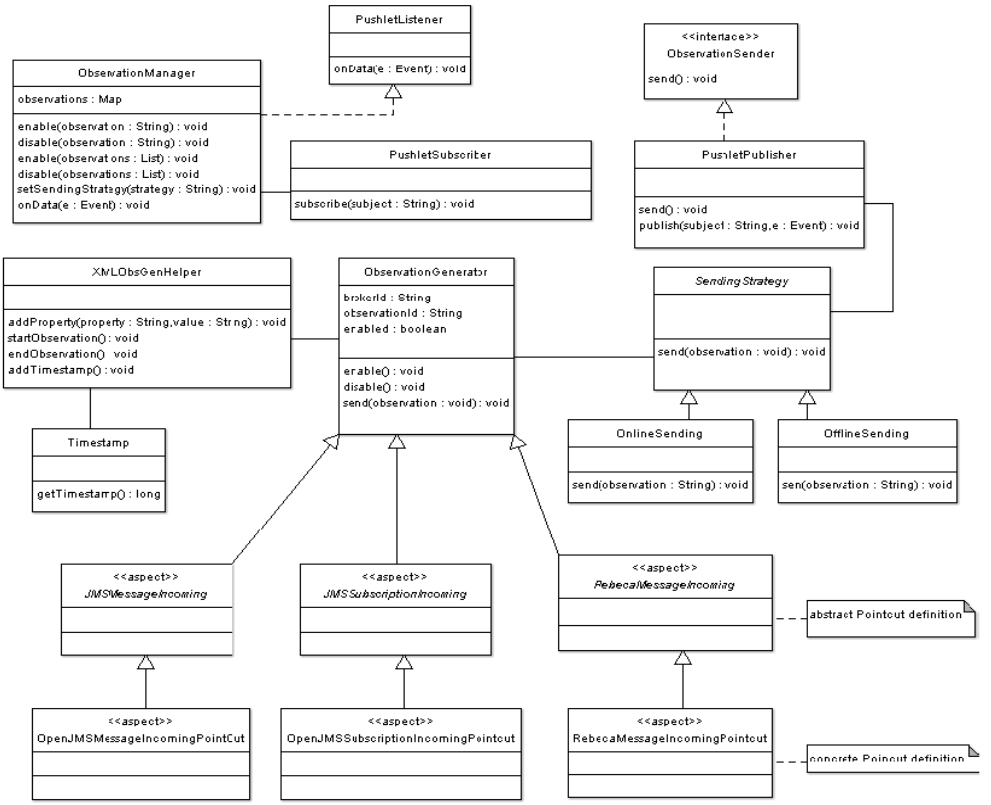

Fig..4. The class diagram for the Behaviour observation layer

5.Conclusions and Further Work

Online analysis. Since the main goal of this work was to enable the online analysis of notification services we started checking it. For this purpose, OpenJMS was taken as the notification service to be observed. To begin making a simple but complete experiment we started defining four different observations: a) when a message enters into the notification service, b) when a message is delivered and left it, c) when the persistence process on a message starts and d) when the persistence process on a message finishes. In order to take these observations the internal organization of the source code was analyzed to find the execution points that corresponds to each of them. Eight aspects were implemented, four abstracts ones for advices and four for concrete pointcuts definitions. Afterwards, the aspects were weaved into OpenJMS code without recompiling it since it is possible to weave using the binary code. The same metrics were defined calculating the elapsed time per message whiting the broker, and periodic

snapshots of the average elapsed time and throughput. A new metric was defined from the observations that observe the start and end times of persistence mechanism in order to get persistence consumption time per message. Finally, we took an open source chart library [11] that have support for HTML Canvas and also SVG. This library provides line, bar and pie charts and they were integrated. These metrics and visualization components were deployed as default configuration and some mappings between them were defined to support online analysis.

In order to check the notification service independence of our solution we took the REBECA notification service to put it under observation. New aspects were developed to generate new observations. The observations taken were a) when a message enters into the broker, b) when a message is delivered and left the broker, c) when the routing mechanism starts and d) when the routing mechanism finishes. The observations a) and b) are the same that were generated in the previous test. They are taken from different aspects but generates the same data. The observations c) and d) for routing mechanism are taken instead the persistence mechanism observed in the previous test. Additionally, in order to check the flexibility to add new visualization techniques, a new third party library of charts was taken. This provides a large set of charts and gauges that are developed with Flash technology. They differs from the first library used, receiving an XML document as input for data to represent.

Taken advantage of the large variety of visualization techniques that it offers, some of the metrics that were visualized before are now graphically represented in different fashion. A new metric that get as measure the routing consumption time was developed from the new observations taken. Metrics definition for elapsed times and throughput remain the same as in the previous one and were reused, new mappings between metrics and visualization components were defined. Figure 5 shows a sequence of screenshots taken during this experiment.. This experiment allows to probe that metrics definitions can be reused to analyze different notification services. Moreover, a new set of visualization components (implemented with different technologies) were integrated without major effort. The same metrics could also were represented by different visualization technique just mapping them to the new ones recently integrated

Conclusions and further developments. Throughout all this work we have introduced a notification service analysis-independent framework that allows online behavior analysis based on streamed observations, metrics and visual representations of them. The utilization of three different kinds of components to observe the behavior *(aspects),* compose streams of observations *(metrics)* and graphically represent them *(visualization techniques*) as well as their loosely-coupled communication through a pub/sub service offer a high flexible framework to enable

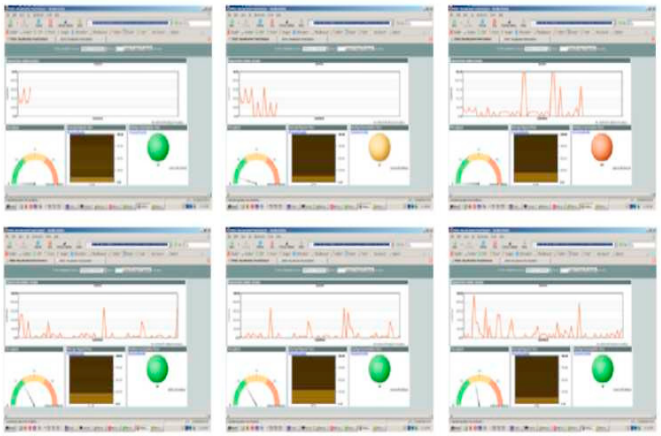

Fig.5. Online analysis using Flash based charts (sequence of six snapshots)

online analysis of notification services. It is possible to get a set of these components and make different

combinations between them to allow high abstraction level analysis on several aspects of the internal behavior. Moreover, new components can be developed and integrated in a very flexible fashion.

By representing the observed behavior as abstract *observations* data and associating its semantic to behavior description instead source code it was possible to get independence of notification services implementations. Additionally, the use of AOP to generate these *observations* allows to avoid an intrusive modification of the source code and let organize in a modular fashion the logic associated to the behavior observation concern. Although the influence on the notification service under observation is a problem that should tend to zero, the designed mechanisms that allow to enable/disable the generation of *observations* and also to vary their sending frequency allow to control this influence.

Although the implementation is a prototype and tests were made using low workload, the use of a pub/sub service as communication model and the loosely-coupled components involved allow to scale the solution to be used in high workload real environments.

Enable the analysis of notification services while running has been the major task of our work . Regarding behavior observation static woven aspects has been written to detect the behavior. An improvement for this task could be to use dynamic weaving or to support a declarative and independent way to describe *observations* and automatically generate the aspects involved letting choose different AOP alternatives.

Metrics and test run definitions rely now on XML documents. It would be helpful to develop several graphical wizards and interfaces upon these descriptions to automatically generate them. This could bring a higher abstraction level to users and offer a complete analysis-oriented application.

The topology and messages flow visualization through an event broker network has not been attached in our implementation. New aspects could be developed to observe the creation of brokers and links and visualization techniques could be integrated to support it.

Finally, since metric definition is associated with observations' semantic and visualization is decoupled from their semantic, new applications domains for the same solution could be explored. Developing aspects over other software systems could enable the same online analysis like for notification services. By observing workflow engines, it could also be possible to get business activity monitoring analysis near real-time. Avoiding aspects and developing other components that subscribe to the activity of a notification service and produce *observations* from messages flow it is also possible to get online analysis on the activity of specific applications that rely on messaging systems.

Acknowledgements

This work was supported by the PNII-IDEI 328/2007-2010, QAF-Quality of Service Aware Frameworks for Networks and Middleware research project within the framework National Research, Development and Innovation Program initiated by The National University Research Council Romania (CNCSIS - UEFISCSU)

References

1. P. Th. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. ACM Computing Surveys, Vol. 35, No. 2, June 2003, pp. 114-131.
2. G. Muhl and L. Fiege. The REBECA Notification Service, 2001. http://www.gkec.informatik.tu-darmstadt.de/rebeca/
3. R. Baldoni and A. Virgillito. Distributed Event Rout ing in Publish/Subscribe Communication Systems: a Survey. Technical Report 15-05, Dipartimento di Informatica e Sistemistica ,2003
4. A. Carzaniga and A. L. Wolf. A Benchmark Suite for Distributed Publish/Subscribe Systems. Technical report, Dept. of Computer Science, University of Colorado, 2002.
5. F. Araujo and L. Rodrigues. On QoS-Aware Publish-Subscribe. In *Proc. of the 2002 Intl. Workshop on Distributed Event-Based Systems*, pages 511–515, 2002.
6. Gianpaolo Cugola, E. Di Nitto, and Alfonso Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. In IEEE Transactions on Software Engineering, 2000
7. P. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In In Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS02), Vienna, Austria, July 2002.
8. P. R. Pietzuch. Hermes: A Scalable Event-Based Middleware. PhD thesis, Computer Laboratory, Queens' College, University of Cambridge, February 2004.

9. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and WG Griswold. An overview of AspectJ. Lecture Notes in Computer Science, 2072:327-355, 2001.
10. Hilsdale, E. and Hugunin, J., Advice Weaving in AspectJ. International Conference on Universit`a di Roma "La Sapienzia", 2005.
11. Robert P. Bosch, Jr. Using Visualization To Understand The Behavior Of Computer Systems, Stanford University, Ph.D. dissertation, August 2001.